Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche
researchportal.unamur.be
University of Namur

THESIS / THÈSE

MASTER IN COMPUTER SCIENCE

Support of Modern Print & Printers Languages by using Filters and Backends

Belhomme, Didier

Award date:
1993

Awarding institution:
University of Namur

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 06. May. 2023

Facultés Universitaires Notre-Dame de la Paix
Institut d'Informatique
Rue Grandgagnage, 21b
B-5000 NAMUR

Support of Modern Print & Printers Languages by using Filters and Backends

Didier Belhomme

Promoteur : Jean Ramaekers

Mémoire présenté en vue
de l'obtention du grade de
Licencié et Maître en Informatique

Année académique 1992 - 1993

Abstract

This thesis covers the field of printing in large-scale computing environments. In order to support the new printing technologies offered today, and to provide the user a reliable interface to the printing environment, new functions have to be included into the operating systems and their subsystems. The function discussed here is about the development of a generic interpreter of printer languages, in order to trace the printer activity to allow it to restart after any problem that could occur. It covers not only the print job in itself, but also all the resources that are part of the printer or that have been loaded into the printer by the print job, and selected during its progress.

Résumé

Ce mémoire couvre le domaine de l'impression dans les environnements informatiques à grande échelle. Afin de supporter les nouvelles technologies offertes aujourd'hui, et pour fournir à l'utilisateur une interface fiable vis à vis de l'environnement d'impression, de nouvelles fonctions doivent être incorporées dans les systèmes d'exploitation et leurs sous-systèmes. La fonction qui fait l'objet de ce travail concerne le développement d'un interpréteur générique de langages de contrôle d'impression, dans le but de suivre l'activité de l'imprimante pour lui permettre de reprendre le travail après qu'un problème, quel qu'il soit, se soit déclaré. Cela ne concerne pas uniquement le travail d'impression lui-même, mais aussi toutes les ressources qui sont intégrées dans l'imprimante ou qui y ont été chargées par le travail d'impression, et sélectionnées durant son traitement.

Acknowledgment

I would like to warmly thanks here :

My promoter, Mr. Jean Ramaekers, and his assistant Mr. Joël Hubin, for the help and the advises they give to me,

Mr. Nikos Piperakis, from Siemens-Nixdorf Software, for allowing me to work in his team,

All the members of the Siemens-Nixdorf Software team SWN14, for their patience and the help they give to me,

All those, at last, I have forgotten.

May they find here the expression of my gratitude.

Introduction

This document is relating to the field of printing. It has been written as thesis to get the title of "Licencié et Maître en Informatique". Basically, it covers the latest printing technologies, available now in large and even small-scale printers. But, one cannot say that the last word has been said. Printing is evolving very fast : new printing technologies are appearing, new applications are developed, needing new, special functionalities to ease the work of user, or to improve performance.

The topics covered in this document are mainly the result of the work done in Siemens-Nixdorf Software in Namur. Consequently, some references are made to products developed by that company. Most of the items discussed here are applicable to any printing environment, but some considerations are very specific to the approach of printing used in the products from Siemens-Nixdorf.

The main problem discussed here is about the restart event. A restart occurs whenever, during the processing of the printing job, the printer is no longer able to process the job and that a special action is needed to complete printing. This action could be automatic (in case, for example, of a temporary loss of connection), or manual (paper loading). When restarting an interrupted job, one should be aware of the context of the printer at the moment problem was detected. This context is essentially, the resources loaded into the printer, and the resources which were selected by the user. These resources include : font, macros, character style, vertical spacing, and so on.

In order to offer a better restart, Siemens-Nixdorf has launched the development of a program analyzing the printer data stream and extracting from it all the data necessary to reset, at the restart, the printer into the exact state it were before the problem. This is the design of this program, called "parser" which is discussed here.

This document is made of four chapters. The first one gives a introduction to modern printers technologies and the languages available to control them. The second chapter is the development of the parser itself, and its integration into the printing subsystems under BS2000, the operating system of the mainframe computers from Siemens-Nixdorf. Chapter three is the detailed specification of the parser. The fourth chapter covers another aspect of printer support : the design of filter translating a printing language into another, allowing a transparent conversion for the user.

Chapter 1
Printers and printing languages

1.1 Introduction

From the very beginning, computers were connected to machines allowing them to keep a written form of data : printers. Following the evolution of processors, printers evolved from typewriter-style printing to large scale printers capable of printing hundred of pages in a minute. This evolution was made possible by continuously increasing the functionalities of printers, what we shall call here : printing language. The objective of such a language is to allow processor to describe, the more accurately possible, what the printing should be. The figure 1.1 shows the logic flow of the data between an application program and the printer, by the way of any communication channel.

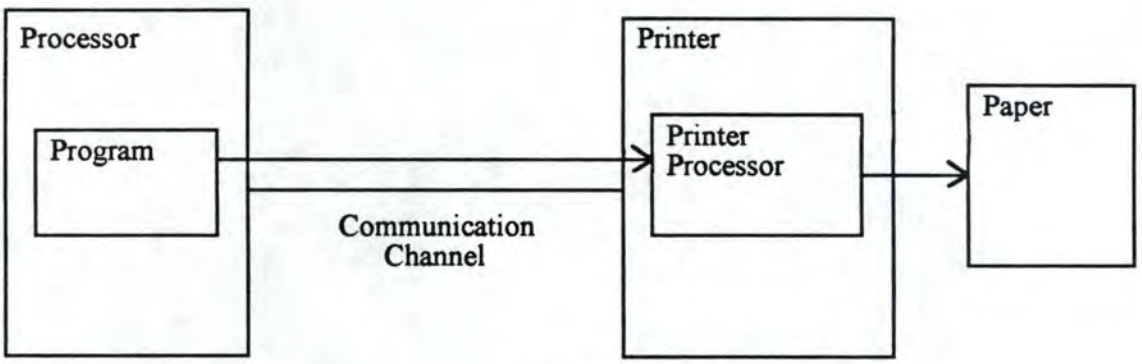


Figure 1.1 - logic flow of data

Generally, a program willing to print some information does not directly generate data in a format directly understandable by the printer. It relies on services available into the operating system, which takes care of all the printing environment. These "Presentation Services" are, for examples, those included a a graphical interface like OSF/Motif. From the program point of view, it only sees standardized primitives, independent of the language used by the printer itself. For now, we will discuss the main differences between traditional line-oriented printers and page-oriented, and the way of controlling them.

1.2 Line-oriented printers

Before the development of laser printers, the high-speed printers were mainly impact printers. These printers used several technologies (drum and chain), but they all had something in common : they printed one line of text at one time. This way of building printers had some limitations : you couldn't have more than one character

spacing (fixed by the drum or chain used, usually 10 characters per inch) and you had no access to special characters if they were not engraved into the chain. Paper advance also offered very few options : the advance was of 1/6 of inch or 1/8, depending on the setup of the printer. Controlling these printers was very simple : every record into the file translated into one line on the paper.

To gain more control over the paper advance, a new concept was introduced : "loop band". That loop was in fact made of paper, with special perforation in it. The advance of the loop was synchronized with the paper and of the same height. For example, for paper of 11 inches, the loop has the same size than the paper, and every time the paper advanced, the loop advanced the same amount. You had the capability to punch the loop to point out special position on the paper, called "channel". There were a limited number of channel in the loop (usually 12). The printer could understand orders like "skip to channel 8", and so it advanced the paper until the perforation corresponding to channel 8 was detected into the loop.

In order for the user to support that functionality, a new format for print files was designed. It used (and still uses) the first character of each line to instruct the printer to skip to a special channel, or to only skip one line, and even no skip at all, which allows the printer to produce special characters by combining two or more other characters.

Printers evolving, the loop was no longer physically present into the printer, but rather kept into a special buffer and loaded before the print job was started.

When, in the '70s, the first laser printers were introduced, they were controlled more or less the same way, in order to remain compatible with the applications written using that method of control. But the non-impact technology permitted new functionalities to appear. Printers were capable of printing different characters on the same line, changing not only the style, but also the pitch of the characters : beginning a line with 10 characters per inch then switching to 15 characters per inch was made possible as soon as there was no physical relationship between the printing mechanisms and the paper. The laser remained the same, whatever the character. To offer these functions to the user, it was necessary to extend the format described previously. The concept of page was introduced, with a special line at the beginning of each page indicating the font to use, whether or not to use the overlay function, and the number of copies of the same page to produce. For switching between characters into the line, it was necessary to offer some kind of "escape sequence" which were signaled by a non-printable character (for example, character 255). But, the line-oriented control of the printer was not changed : the loop remains, and it was impossible to modify a line in the top of the page when printing the bottom. As soon as the order to advance the paper was received, it was done. That way of controlling printers is still in use, for it is well-suited for large volume of printing, not requiring a very high quality (listings, massive amount of invoices or bills). These printers generally do not offer "all points addressable" feature, except for special needs like overlays. Since the format of these printers is still widely

used, it is necessary to offer a way to support that format on printers with another technology, by using filters which should completely be transparent for the user. The way of building such support is discussed later in this document.

1.3 Page-oriented printers

The name of "page printer" applies to a printer which allows a program to work within all the page at a time. That was a requirement to allow programs to use new technologies available with laser printers, like font management, electronic overlays and graphics. These printers are, for the most, "All Points Addressable", that is, a program can address each element of the printing, the pixel, like it can do with a graphical display. The "All Points Addressable" characteristic is an important one, as it is required for graphics integration.

The evolution from line-oriented printers was essential. The line-oriented printers (mostly impact printers) allows the program to send one line of printing at a time. As soon as this line is physically printed on paper, and the paper has advanced to the next line, there is no way for the program to modify the line. The program should be aware of all the data to print on a line. For example, imagine a document with 2 columns. For printing the first line of a page, the program should build itself all the first column in memory, to detect the text which should go in the first line of the second column. In fact, in that case, the program has to do the job which is currently done by the printer, when it has a page-oriented language of course.

1.3.1 General description

Page-oriented language are matched with page-oriented printers. These languages have been viewed as extensions to a text stream (escape sequences like the ones used in PCL), or as completely new format (PostScript language). All these languages have the characteristic not to directly print the data as soon as it is received, but to rather build an image, in memory, of the page that will be printed. This is the job of the Raster Image Processor. It interprets the data received from the processor, converts it into a pixel constellation, and finally, use this description to control the mechanics of the printer (laser or LED). The figure 1.2 shows the inside of such a printer.

printed. Such technology is often used when printing massive amounts of paper (from 100.000 to 1.000.000 pages).

These resources can be resident in the printer, or loaded on demand by an application program. It is important to note here that the loading of resources should always be done before the print job can begin. In most printers, reference to an unknown resource leads to the stop of the printer, with an error code. In that case, the application program should be informed of the error, and it is to its responsibility to process the error and restart the printer. Up to now, when we talked about "application program", it refers to any program running into the processor. Later in this document, we will describe the software organization of the operating system relative to printing control.

The way the resource are loaded into the printer depends on the printer language. For example, PostScript use a kind of font called "outline fonts", whereas IPDS uses "bitmap fonts". Being quite different, these two kinds of fonts requires different loading procedure. Some languages support concurrently outline and bitmap fonts (PCL 5).

Bitmap fonts are older. Each character is described pixel by pixel, in one size and style. Figure 1.3 shows the way the letter "a" can be described in a bitmap font.

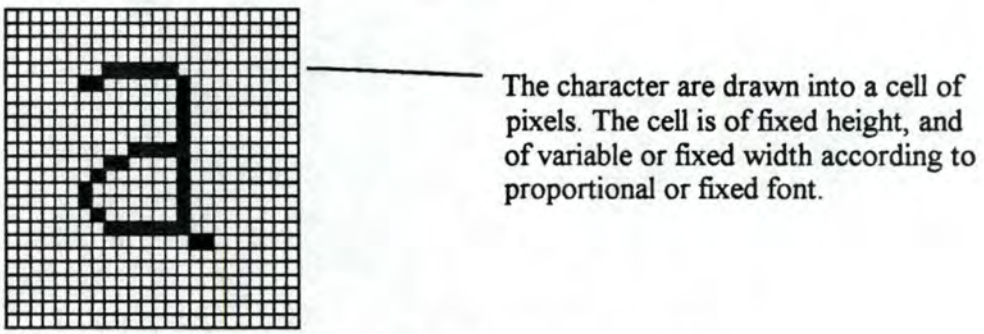


Figure 1.3 - cell of a bitmap font

Outline fonts were popularized by PostScript. These fonts give the outlook of the characters with a combination of vectors, describing the font in a manner independent of the size. Figure 1.4 shows how the letter "**a**" is described in an outline font.

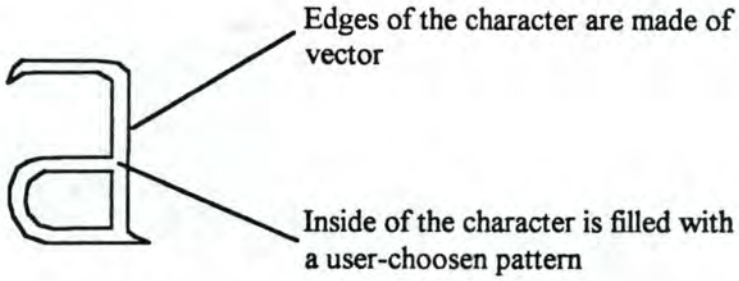


Figure 1.4 - outline font

Being quite different, these two kinds of font are loaded into a printer by different techniques. In the case of bitmap fonts, we will have to manage long sequence of binary data, and with outline fonts, a great number of vector description.

1.4 From computer to printer

On a first view, the method used to send the data to the printer can be considered to not infer on the control of the printer. But in fact, it's not always true. Some communication environments don't allow long sequences of characters : they cut them in the middle, and add some control characters of they own, leading to an incorrect interpretation of the data stream by the printer. Let's see the main interface used today to communicate with printers.

1.4.1 Asynchronous serial interface

This interface is widely used by terminals. It offers character-oriented communication. The code used is nearly always ASCII code, which allows binary transfer of data. This interface allows bidirectional transfer, allowing the printer to send an acknowledgment to the computer and permitting pacing control (XON/XOFF protocol). This interface is very efficient between an computer and a printer, but when used between two computer, does no usually offer transparent transmission. For example, the control characters could be incorrectly interpreted if they are embedded into binary sequences sent to the printer. If no transparent mode is offered by the protocol, it is usually impossible to send binary data to printer, that is, no graphic nor font loading.

In the case of a printer directly connected to a computer, the printer is able to distinguish between the data flow in itself and the protocol used to communicate with the computer. When embedded into a binary sequence (that is, a sequence containing a binary part defining graphic or font), the printer doesn't react to control characters, but rather process them as appropriate in the data stream. The problem is when the communication environment is not aware of the kind of data it has to support : in that case, special characters belonging to the protocol used in the communication should be forbidden into the data stream, or a special sequence of character should be used. For example, the protocol can realize what we can call a

"byte stuffing", by adding a special character before any character to sent to the printer and belonging to the protocol. But of course, the protocol implemented in the printer should be aware of that way of doing, and should remove the character added to not interpret it as being part of the data stream.

Fortunately, there are printing languages that don't require transparent transmission, like PostScript. In that case, graphics and font loading is available on nearly all communication methods.

1.4.2 HDLC synchronous interface

This interface offers a block-mode transparent transmission. The printer controller should only remove the header and trailer of each block of information, and only decode the data part. This interface bidirectional transmission, allowing the printer to acknowledge the computer.

HDLC is widely used to connect peripherals to a computer, in a wide-area network. This protocol is highly reliable, and its block-mode of operation offers an high throughput. But it is a rather complex protocol, needing synchronous operation, and it is seldom used in the field of personal computers.

1.4.3 Centronics Parallel interface

This is the interface used on personal computers. The interface send one byte of information at a time, with parity control. But this interface is often mono-directional, with only a "paper out" signal. Few information can be used when printer signals an error : that can be anything, from paper out to buffer full to font loading error. Recently, a new norm for parallel interface was introduced, giving bidirectional transfer and so the capability to better control the printer. Error reporting is much better. The interface is of course transparent, but cable is very short : a few meters only.

1.4.4 Channel interface

The channel interface was designed to support a large number of peripherals in the field of mainframe environment. It is designed for high-speed transfer (4.5 MB per second) and to offer simultaneous bidirectional transfer. It is transparent, allowing full control of the printer.

Channel-attached printers are often very fast ones. These printers are mainly used for large printing jobs, and they are used in large computer environments. Being bidirectional, the channel allows the printer to make detailed report about the completion of job, or to inform when an error occurs.

1.4.5 Local Area Network interface

A new type of interface, the local area network attachment is becoming very interesting when a printer should be shared by several computers. As usual in a local area network, a printer connected to it works in a "peer to peer" approach with all the devices connected to the network. This leads to the fact that a higher-level protocol should be implemented in such printers, not only offering transport functions, but also managing access conflicts between two computers. These protocol are not yet standardized. Currently, printers offering direct attachment to a LAN usually come with protocol from Novell Netware, Microsoft Lan Manager and TCP/IP lpd and lpr client/server approach. Decoding these protocol is no easy task, and in fact these printers often offer functionalities from a personal computer, with hard disk and memory, to implement the protocol used. The main advantage of sharing a printer through a LAN is that there is no "master" in that approach : as long as the printer is functioning, all the computers on the network can gain access to it, and have not to rely on a special server for printing purposes.

1.5 Printing languages

For the purpose of that work, we will discuss here the pro's and con's from three main printer languages : IPDS, PCL and PostScript. These languages have been chosen because they are all typical of a printing environment. IPDS is the "de facto" standard for large-scale IBM laser printer. Some other manufacturers, willing to develop printers for the IBM environment, are forced to offer this language to be compatible with IBM printers. PCL is the language mainly used by the Personal Computers, where it is used to control small laser printers, but also other non-impact printers and even dot-matrix printers. PostScript is the language generally used when high-quality printout is needed : it is also used by typesetters for high-resolution flashing for professional printing.

1.5.1 IPDS

IPDS (Intelligent Printer Data Stream) is a printer control language developed by IBM. It provides an attachment-independant interface for controlling and managing all-points addressable (APA) printers. That kind of printers allows the printing of pages containing an architecturally unlimited mixture of different data types : text, image, graphics, and bar code. The difference between image and graphics is visually unnoticeable : both look the same for the user. The difference, for the printer, is that image are described pixel by pixel (bitmap), whereas graphics uses special commands, usually under the form of vector. The bar code are so often use by printers that IPDS includes special commands for printing them. Besides a printing language, IPDS incorporates the following features :

- Different applications create source data (graphics, image, bar code and text) which may be independent of one another. IPDS can merge these data streams at print time, to produce a page combining these elements.

- IPDS offers a high level of independence from the lower levels used for the transport of data to the printer (that is, network environment). The IPDS printers can be connected by channels, local area networks, wide area networks to the processor. The only requirement is that the transport is able to carry transparent data.

- The data stream of IPDS is made of "structured fields". These structured fields describe the presentation of the page and provide the following :

  - Dynamic management of resident fonts and downloaded resources (overlays, page segments, loaded fonts).

  - Control of device functions (duplexing, media-bin selection, output options).

  - Handling of exceptions (which is the main point in this context).

- IPDS provides an acknowledgment protocol, imbedded in the data stream. This protocol allows the computer to synchronize with the printer, and to return detailed information about the error.

1.5.2 PCL

PCL (Printer Control Language) was defined by Hewlett-Packard, for integration into its first laser printer launched in the Personal Computer market. It was designed to be simple and efficient, to maximize the throughput of the printer. One important characteristic of PCL is that it is designed as an extension of the simple data stream which was available with the Teletype-like terminals. Using ASCII, this data stream included control characters (like carriage return, line feed, vertical tabulation and so on) which were used by the terminal to arrange the output on paper. These control characters are so common today that they can be considered as a standard. The ASCII code includes these control codes, and nearly all printers understand them. That is, is the simplest form, any PCL printer can be seen as plain-old Teletype terminal printer. This is very different from PostScript or IPDS : in both case, the printer only understand its language, and printing a simple text file needs to "encapsulate" it into PostScript or IPDS commands.

The power of PCL is delivered by the usage of escape sequences. These sequences of characters begin with the ESC character (character 27 in the ASCII code), followed by parameters and characters describing the function of the language to be activated. All the escape sequences are variable-length.

PCL offers (up to now) 5 levels of control.

- PCL 1 : print and space functionality.
- PCL 2 : Electronic Data Processing / Transaction functionality.
- PCL 3 : Office Word Processing functionality.
- PCL 4 : Page Formatting functionality. It offers all the functionalities of PCL 3, plus the graphics capability in a page-oriented approach.
- PCL 5 : Office Publishing functionality. This level adds the HP/GL plotter language and vector fonts.

Pre-PCL 4 are today mainly historic. All the laser printers available today support PCL 4 and the vast majority offers PCL 5 language.

1.5.3 Postscript

The best definition of PostScript remains the one given by Adobe Systems (creator of the language), as printed in the Reference Manual [1].

"The PostScript Language is a simple interpretive programming language with powerful graphics capabilities. Its primary application is to describe the appearance of text, graphical shapes, and sampled images on printed or displayed pages. A program in this language can communicate a description of a document from a composition system to a printing system or control the appearance of text and graphics on a display. The description is high level and device-independant."

One of the main advantages of PostScript is that the data stream is text-only, that is there is no need for binary data : if you can send text with only some special characters (backslash, brace and curly brace), it's enough to send PostScript code.

A print job in PostScript is rather a program for the printer, and it is processed like any other program is interpreted when using an interpreted language (like BASIC). For example, the following program

```
40 50 moveto (ABC) show
showpage
```

produce a page with ABC printed at position 40,50 in the coordinate system used by PostScript.

Chapter 2
Integration of the parser in RSO

2.1 Introduction

In this chapter, we will describe the design of the parser to be included into the Remote Spool Output (RSO) subsystem under BS2000. First, we will present the RSO subsystem and its architecture, what it does and how it is related with others subsystems under BS2000. Then what will the parser do and how the parser will be integrated into RSO are discussed. The architecture of the parser is next presented, and finally, the modifications to be done to the parser for integration into the new SPOOLPCL subsystem are presented.

The parser has been developed in order to offer a more reliable printing service to the user. Its job is to "do its best" to track the printing activity in order to be able to restart correctly after a problem occurred during the printing job.

2.2 Architecture of the RSO subsystem under BS2000

The parser has been designed to be integrated, in a first step, into the Remote Spool Output (RSO) subsystem. But what are the functionalities of RSO ? As any subsystem under BS2000, RSO has the following characteristics :

- Strong integration with the operating system. RSO offers an extension to the classic command for starting a print job : the ability to print on devices not connected to the computer through a channel (these directly-attached printers are managed by another subsystem, SPOOL, which is mandatory in any BS2000 implementation).
- Loading and unloading on demand. Like the other subsystems, RSO is managed by commands in the operating system BS2000. These commands start the subsystem (START-SUBSYSTEM), suspend the activity of the subsystem (HOLD-SUBSYSTEM) and stop the subsystem (STOP-SUBSYSTEM), freeing the memory used by it and making unavailable the commands that require its presence.

For the user, RSO offers the following functions :

- Printing of files on remotely connected printers. The different connection types supported by RSO are illustrated in figure 2.1. They range from direct HDLC connections to LAN connections to emulation programs.
- Dynamic start and stop of printer activity. By that mean, the user can control the dispatch of the jobs on the printers. Starting a printer means beginning to

dispatch jobs waiting on queue. When stopping a printer, the user has the choice to cancel the job currently printing, or to wait its completion before stopping printer. Stopping the printer does no cancel the jobs waiting for it.

- Control of jobs directed to a printer. The user has the privilege to cancel a job and to check the printers queue.

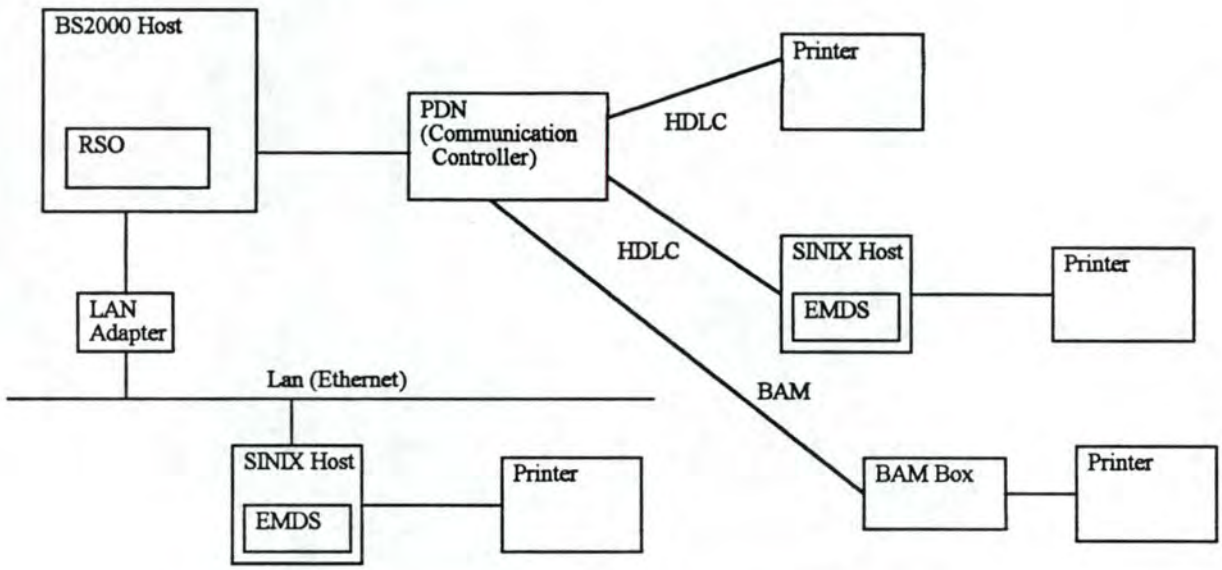


Figure 2.1 - Some connections supported by RSO

The connections used by RSO are numerous. They range from HDLC lines to LAN connections to an emulation program running under SINIX (EMDS), through BAM interface with a conversion box (the BAM Box. BAM is a Siemens-Nixdorf interface used mainly for terminals). The PDN (communication controller) is used to control the network environment, which of course is not dedicated to printer control. The whole network is controlled by the TRANSDATA network protocol.

The functionalities of RSO are activated by the standard PRINT-FILE command, but only when the device specified with it is among those managed by RSO. As RSO uses some function of the SPOOL subsystem, this one should be loaded before RSO itself. RSO uses the administrative task of SPOOL.

The task structure, simplified, of SPOOL/RSO is given into figure 2.2. Let's discuss now the function of each task appearing in this figure.

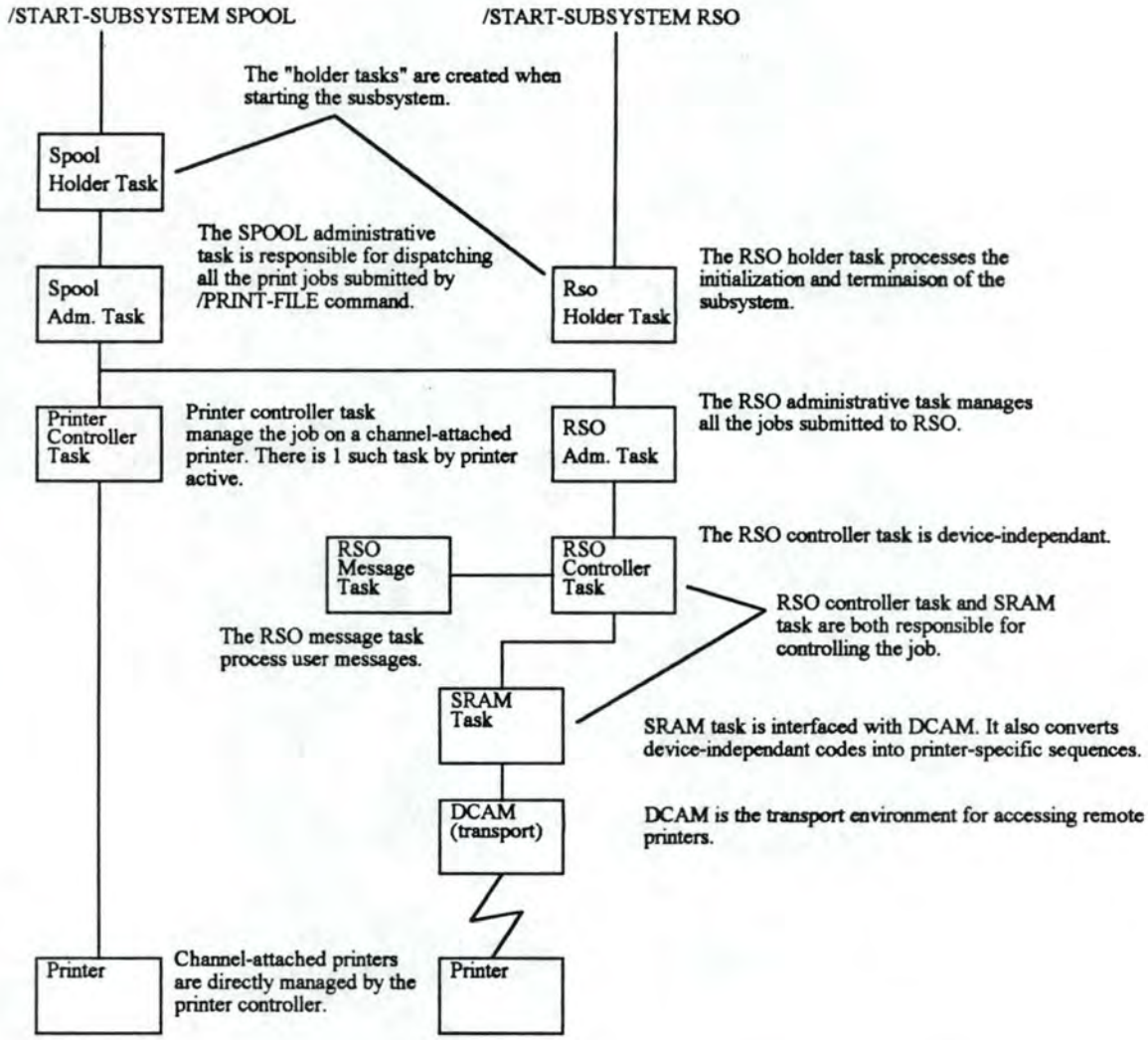


Figure 2.2 - Task structure of SPOOL/RSO

First, as soon as any subsystem is started under BS2000, a special task, called "holder task" is created. It is up to its responsibility to manage all the subsystem. This holder task is responsible for at least two major events in the activity of the subsystem :

- Initialization, when the subsystem has to gain access to all resources it needs for working, and must start the all tasks necessary for its activity.
- Termination, when the subsystem has to do the cleanup of all its components (like closing opened files, freeing memory, ...).

In the case of the SPOOL subsystem, the holder task becomes the administrative task. The administrative task is the one responsible for all processing of a print job before it is actually processed by the printer. The main function of this task is to manage the printer queue, common to all print jobs, and stored into a special file,

the EQUISAM file. The fact that all the queue resides in a file on disk, allows the print jobs not completed when the system is stopped, to be restarted at the power on of the system and the initialization of the subsystem.

In case of RSO, the holder task does nearly nothing. In order to offer the same user interface for both SPOOL and RSO, all the administration of the jobs is done by the SPOOL administrative task. On initialization, the RSO holder task checks whether SPOOL subsystem is operating. If it is not, it waits until the SPOOL becomes available. Then, the holder task notify the SPOOL administrative task to launch the RSO administrative task. The holder task is then waiting, and will stay in this condition until the subsystem will be stopped (/STOP-SUBSYSTEM command), in case it will notify all its components to stop activity and terminate execution.

The administrative task of RSO is responsible for the management of the jobs directed to it from the SPOOL administrative task. Among other things, the RSO administrative task is responsible for selecting the printer controller to use for the job, according to the target printer specified in the /PRINT-FILE command. The printer controller is the last task in the processing of a print job. It is the task directly connected with the printer. In the case of RSO, the printer controller is in permanent dialog with the SRAM task, which is responsible for network access. Why this separation in two different tasks in RSO ? Because of the privilege level of execution. BS2000 has four privilege levels, which correspond to hardware protection levels (at least, until recently). These privilege levels, numbered from P1 to P4, are associated with the opportunity for a program to have access to hardware resources or not. Since the printer controller should have access to the printer at the hardware level, it has to execute on the P2 level. But the communication system used by RSO (DCAM) is only available from P1 level (the user level, with minimum privilege). Thus, a communication mechanism has been developed between the printer controller in itself, and the SRAM task, associated with it in a 1-to-1 relationship, to allow the printer controller to send data to the printer.

RSO allows more than one controller to be activated. In the SPOOL subsystem, there is one printer controller for each printed activated. In RSO, the high number of printer and the low amount of data to sent to printers controlled make this approach difficult to implement and unnecessary. A printer controller has thus the ability to control a large number of different printers.

No printer controller is created at the start of the subsystem. In fact, it is only when processing a command to start the output to a printer (/START-PRINTER-OUTPUT) that the administrative task of RSO creates a controller and its SRAM task. It is only when processing a command to stop the output to the printer (/STOP-PRINTER-OUTPUT) for the last and only printer managed by that controller that the controller and its SRAM task are destroyed. The /STOP-PRINTER-OUTPUT command has an option that indicate the behavior of RSO when a job is being processed by that printer. When this option (FORCE=YES/NO) is activated, the job

currently being printed is canceled, otherwise the job is completed until its end, then the controller and its SRAM are destroyed.

The data sent from the controller to the SRAM task is device-independant. It is the SRAM job to convert special codes into specific sequences for a particular printer.

In BS2000, a print job is always relative to a user file. The /PRINT-FILE command does not work with internal pipes or other things : to be printed, any data has to go to a file, even a temporary one. Thus, it is only when the job is actually send to a printer that the user data is accessed.

2.3 User interaction with RSO

Besides starting and stopping the subsystem, the user interface offers the following functions :

- Managing device information. This include defining printer type, connection type and so on, in a word giving RSO all the parameters needed for accessing the printer.
- Creating resources. These resources are not those directly available in the printer, nor those loaded by the user job, but rather those specified with the /PRINT-FILE command. These resources includes :
  - Form description. The form describe the physical paper loaded into the printer : vertical and horizontal sizes.
  - Character set. In opposition with the SPOOL- controlled printers, where the character sets used by a job have to be loaded into the printer before the job can start, RSO does not load character sets in any printers. But, since the user can ask for a special character set, RSO tries to do its best by selecting the character set, available into the printer, which match best the character asked for by the user.
  - Loops. The loop describes the vertical advance of the paper for each printed line.

All these resource definitions are managed by the program RSOSERVE and SPSERVE. The resources are stored into special files, here called "parameters files".

2.4 Functionalities of the parser

We intend now to give an "abstract" description of the parser, that is "what the parser will have to do, and why" rather than "How", which will be discussed in chapter 3. From the elements given in chapter 1, we saw that printing languages are

numerous and quite different. In fact, even if we discussed only three of these languages, they have to be seen here as examples from the three environment where they were developed. Nearly each printer has special capabilities which are not included in the standard language. Currently, RSO does not offer a high level of restart. During processing of a job, the printer controller creates a checkpoint entry in a table for each buffer sent to SRAM. Each of these buffers correspond to a page to be printed, and has reference to the record number of the file which correspond to the beginning of the page. In case of an error notified by SRAM, the controller send again the buffer for which it has not yet received positive notification. No information about the printer environment is saved in these buffer.

In the SRAM task, some features are saved when the conversion process from device-independant to printer-specific code is done. But this conversion process is only done for RSO-managed features, and not those selected by the user file itself.

The objectives of the parser is to improve that. By analyzing all the data sent to the printer, the parser will be able to save all the information, into the data stream, which relate to resource loading and selection.

2.5 Integration of the parser in RSO

From the description above, it seems that the only place for the parser is inside the SRAM task. Effectively, it does not make any sense to try to analyze the data before it is actually converted for a special printer : the user-specific features inside the data stream will not be caught in that case. The parser should be the last to process the data to sent to the printer, as nothing can modify the data after it has completed its job.

The parser will be seen as an extension of the SRAM task. It will offer functionalities not only to build the restart environment, but also to offer to SRAM several functions in order to insure a better control of the printer. For example, up to now RSO does not allow binary sequences to be sent to the printer : due to the network environment, these sequence are split into several part when they are too long for the network. This splitting adds some control characters, belonging to the network protocol, which cause the printer to incorrectly interpret the data stream. The parser will catch that situation, by detecting the length of the sequences before they are actually sent to the printer. So, SRAM will be able to split the buffer between the long binary sequences, ensuring that the printer will never receive sequences with the added characters in the middle. Another function of the parser is relative to the EBCDIC to ASCII conversion. The Siemens-Nixdorf mainframes use EBCDIC as internal code. Some printers connected to RSO are ASCII printers. If the conversion is perfect when processing text or control characters that are in the EBCDIC code, the problem is when sending binary data to the printer. For example, a bitmap image can be send as a sequence of 0 and 1, the 0 corresponding to blank dots and the 1 to black ones. That binary data should not be converted from

EBCDIC to ASCII. Up to now, RSO does not allow this kind of sequences : the data is converted from EBCDIC to ASCII on an one-to-one relationship.

To resume, the parser will implement all the functions that are device-specific, in offering an device-independant interface. This interface will have to offer the following functions :

- Parsing a buffer containing data to be sent to printer, with building a "restart buffer" containing the data to sent for rebuilding the context of the printer.
- Determining the length of a particular sequence in the data stream, in order to allow SRAM to split the block of data in several buffers for the network environment.
- Signaling to SRAM the part of the buffer for which the EBCDIC to ASCII conversion should be turned off.

2.6 Architecture of the parser

The architecture presented here is a summary of the job completed for Siemens-Nixdorf and given in the chapter 3.

The parser designed for RSO should comply with the logic of the SRAM task. We choose to build the parser in an event-oriented approach. For that, the parser has to manage the following events :

- Starting of the subsystem. At that moment, the parser should initialize all the data structure that will be shared between all the occurrences of the parser. This global data structure will be integrated into the global memory already used by the SRAM task for its job.
- Starting a printer. When starting a printer, the parser should allocate memory for its work. This memory will receive the saved data from the data stream.
- Dispatching a new job. At that time, the parser should initialize the restart buffer (the buffer containing the data to sent to the printer to reload the context) to reflect the default state of the printer.
- Processing a buffer. The buffer received by the parser contains the data to sent to the printer. After completion, the parser returns the restart buffer filled. This buffer will be stored by SRAM along with the checkpoint information.
- Stopping a printer. When stopping a printer, the parser can release the memory allocated to that printer.

When stopping the subsystem, the parser has no function to implement : releasing its memory is done by the subsystem itself.

2.6.1 Data structures used by the parser

The figure 2.3 shows the data structure used by SRAM and the parser.

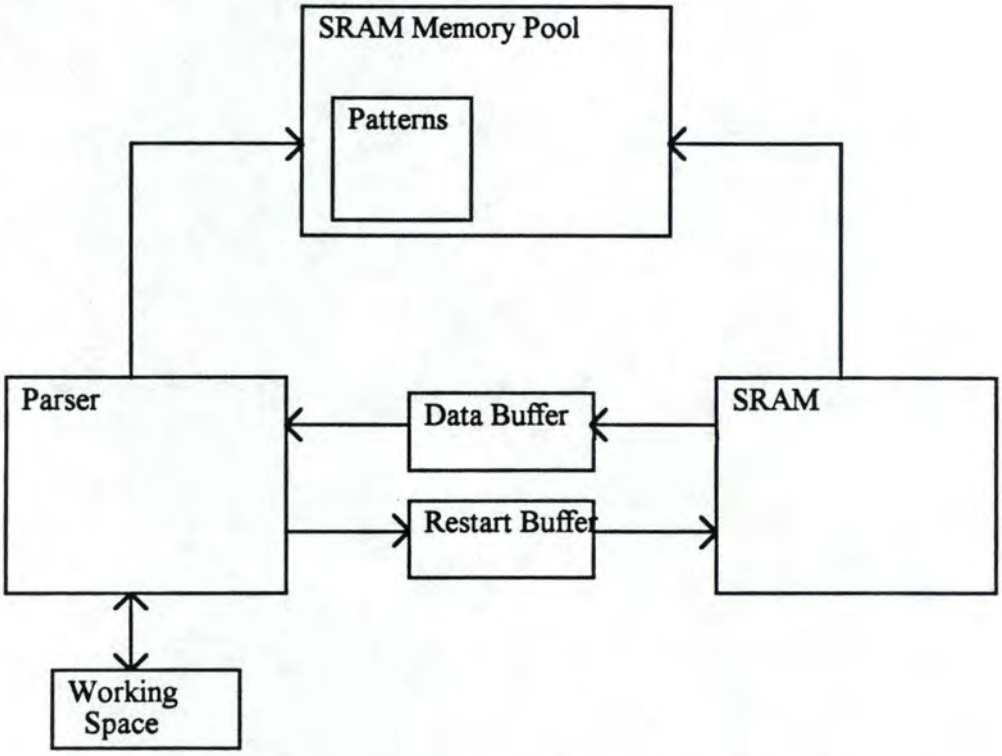


Figure 2.3 - Data structures of SRAM and the parser

The SRAM memory pool contains all the information needed for SRAM and the parser to complete their functions. This memory pool is created at subsystem initialization, and is accessed for reading only by all instances of SRAM and the parser. There is only one copy of this memory pool. It is accessed by the parser as shared memory.

The data buffer is given to the parser by SRAM. It contains the data to sent to the printer.

The restart buffer is built by the parser at the end of the parsing time. This restart buffer is saved by SRAM along with the checkpoint information.

The working space is allocated by the parser for each started device. It is used for internal tasks.

2.6.2 Processing the start and stop of the subsystem

At the start of the subsystem, a function of the parser is responsible for initializing the part of the memory pool in use by the parser. This part contains all the "patterns", that is, the description of all the sequences to be detected in the data stream. Along with the pattern, there is a description of the actions to be done by the parser when detecting that sequence. At the stop of the subsystem, the memory pool is released by the cleanup function.

2.6.3 Processing the start and stop of devices

At the start of device, the parser is notified to allocate private memory in order to be able to process the buffers of data directed to that printer. That private memory is used during the processing of a buffer from SRAM. It contains all the data saved by the parser during processing of a buffer. When can say that this memory is a real "backup" of the actual state of the printer. All the features that were selected into the printer by the print job, are also present in this buffer.

2.6.4 Initialization and termination of a print job

When initializing a new print job, the SRAM task notify the parser to reset the restart buffer, that is to cleanup the private memory to reflect the fact that no feature of the printer has been selected before the job is dispatched. On termination of a print job, no action is done by the parser : the reset is performed at the start of a new job.

2.6.5 Processing of a buffer from SRAM

This is the main function of the parser. Its job is to analyze all the buffer given by SRAM, and try to match the buffer with the patterns that are defined for the type of printer selected.

2.6.5.1Kind of sequences to save

The sequences of data that are to be detected by the parser are of three kinds :

- Toggle functions
- Choice functions
- List functions

The toggle functions : this is the case when selecting underscore characters, bold, italic and so on. These sequences are of the kind "set/reset" : when the sequence corresponding to the "set" function is present in the data stream, that sequence should be saved. But when the "reset" sequence is detected, the effect should not to save the sequence of reset, but rather deleting the "set"sequence corresponding : it is no use to send to the printer both the sequences that set then reset the function.

Example : suppose we have the following data to print :

```
This is text to print with <set underscore>underscored
words<reset underscore> in the middle of the sentence.
```

When printed, the sentence will look like :

This is text to print with underscored words in the middle of the sentence.

During analyzing, the parser will first catch the <set underscore> sequence, which will be saved. Later, when encountering <reset underscore> sequence, the parser should delete the <set underscore> sequence rather than saving the <reset underscore>.

Choice functions are those that select an option among several. Sequences selecting character font are of that kind. When processing such sequences, the parser should replace the sequence of the same kind that may be saved previously. If no sequence of that kind has been saved, a new entry should be created in the restart buffer.

The following is an example :

```
This text is in the default font then<select Times> switch to
Times.
```

which gives the following result :

This text is in the default font then switch to Times.

When processing the <select Times> sequence, a new entry is to be created into the restart buffer.

At last, the list sequences are those made of a variable number of elements. Sequences setting tabs are of that kind. These sequences should be saved with all their parameters when detected. The sequence which resets to default should delete the saved sequence.

From all these kinds of sequences, it leads to the conclusion that when the parser detects a sequence, there is need for a way of describing the actions to be completed. It is not only to instruct the parser to save the sequence, but also reset other sequences.

2.6.5.2 Logic flow of the parsing

The foundation of the parser was a similar job done in the Xprint product under Sinix. The reason for that was reusability. The logic flow for the parsing itself is rather simple : each pattern for the device is considered, then a processing is done to try to unify the pattern and the data in the buffer. If this unification is successful, then the actions associated with that pattern is interpreted. If the action does not terminate by indicating to stop the parsing, parsing is continued with the data following the recognized sequence, until the end of the buffer.

After that, the restart buffer is filled with the sequences saved into the internal memory of the parser. The sequence are placed into the restart buffer in the order they were detected into the data stream, in order to take into account dependencies that can exist between sequences. For example, PCL requires that some sequences be sent in a special order. If the user file does not respect this order, it is not to the parser to correct this error.

The information returned to SRAM by the parser is :

- A code indicating why the parser has stopped the parsing. That can be normal completion (end of the buffer), internal error, indication of the need to stop EBCDIC to ASCII conversion or indication of the processing of the function to detect the length of a special escape sequence.
- The restart buffer itself, ready to be sent to the printer and thus formatted according to the rules for the specific printer. Remember that outside the parser, all function are device-independent.
- The displacement in the buffer were the parser has stopped the job. This information is also used to tell the SRAM task were EBCDIC to ASCII conversion should stop.
- Length information when appropriate (according to the function of the parser selected by the SRAM task).

2.6.5.3Describing the parser action

For these reasons, an "action language" was introduced. This language is made of codes, each indicating a special function to be executed by the parser. This language is described with much details in the appendix. A small "program" in this language is thus associated with each pattern. This program is interpreted by the parser as soon as the pattern is matched with a sequence in the data stream.

2.6.5.4 Performance consideration

A word should be said here about performance. The parser design has largely been inspired from a similar implementation in the Xprint product from Siemens-Nixdorf. In that context, a performance degradation of about 10 percents has been noticed when using the parser. On should be aware that the job of the parser should be done in about 3 ms for one line ! As a matter of fact, it is the time a 20.000 lines per minute printer takes for printing one line. Thus, if we consider a buffer of one page (about 2KB) of data, it has to be processed in about 250 ms. This objective should imperatively be reached for continuous-form printers : it is not authorized to stop such printers at each page ! This would lead to mechanical stress leading to fast degradation of the printer. Given the performance of the mainframe used for RSO, this should not be a problem.

2.6.5.5 Problem handling

Although great attention has been ported to error that can occur during the subsystem activity, it is not possible to make assumptions about the content of the user file. If too much resources are selected by the user, the parser could run "out of space" to build the restart buffer. In such case, the parser notify SRAM of a problem, stop itself working (it is no use to go on in such a case : having lost some

resources information makes the restart useless if no complete) but does not force the print job to stop : if no problem occurs during the job, and so no restart necessary, it is not acceptable to penalize the user. In such a case, a message should be printed at the end of the job, indicating that the user file asked too much resources.

2.6.6 Processing the restart for a printer

Detecting and processing the restart of a printer is done by the RSO controller. Using checkpoint information, the controller is able to re-read information of the user file, then send the restart buffer saved along with the checkpoint. This restart buffer follows the normal way of processing : SRAM with parser analysis, then transport method. After that, the controller send again data from the user file, and continues the job. The parser is not directly informed of a restart, since all its job has been done during processing of the buffer.

2.7 Porting of the parser to SPOOLPCL

The SPOOLPCL subsystem is a new subsystem from Siemens-Nixdorf developed in order to support new printers, using the PCL language, and connected to the computer through a channel. Thus, support of such printers cannot be integrated in RSO (remember that RSO is not responsible for channel-attached printers). This subsystem is still under design, and will be available along with the printers.

2.7.1 Functions of the SPOOLPCL subsystem

As SPOOL, SPOOLPCL will only control the channel-attached printers. These printers are PCL devices, with High Performance printers capabilities for compatibility reasons. Its initialization and termination is similar to the SPOOL subsystem. The parser will be integrated into SPOOLPCL in order to manage the restart environment.

2.7.2 Integration of the parser in SPOOLPCL

Being device-independent, the parser has been designed to support PCL language as well. In this context, however, the only language it will have to support is PCL. Thus, the pattern defined will only be those from the PCL language.

The main difference between the parser under RSO and SPOOLPCL is that the parameters file for SPOOLPCL will be structured differently from the parameters file of RSO. The module loading the sequence should thus be modified according.

2.8 Perspectives for parser evolution

Up to now, RSO subsystem mainly support printers with the PCL language, or language derived from PCL (that is, language using "escape sequences" in a standardized data stream, whether EBCDIC or ASCII). There is currently no direct support for IPDS or PostScript. In the case of PostScript, for example, nothing prevents the user to sent PostScript code (text) to a printer supporting that language. But in that case, RSO is not able to rebuild the printing context in case of a problem : it is the user's responsibility to check himself the problem, and choose to go on or to cancel the job and restart a new one. Nevertheless, the parser has been designed to be adapted for PostScript and IPDS environment.

As long as the printing language used can be matched to the pattern which are the basis for the parser, it is able to build the restart environment. For example, a PostScript sentence like :

```
/Times-Roman findfont 24 scalefont setfont
```

which select the Times Roman font in 24 points size, could be matched with a pattern with the following description :

```
%s findfont %d scalefont setfont
```

and associated with an action string to save this sequence in the restart buffer.

The same is true for IPDS : the binary format of IPDS makes it less readable, but the parser is not restricted to text : binary values can be saved by using character representation. The parser is thus adaptable to other printing contexts. Its design is flexible enough to allow it to be ported to other printing environments. The parser, as already said, will be ported to the SPOOLPCL subsystem, besides being present in the RSO environment.

Chapter 3
Detailed design of the parser

3.1 Introduction

In this chapter, the following aspects of the parser design are adressed :

- Control. The control describes the logic flow of the function. The main structure is presented, but not all the details of the implementation.
- Data. The data structures used by the function, and how they are acessed (read, read/write).
- Assumptions. This is the precondition that must be true before execution of the function, in order for it to execute correctly.
- Performance. The performance consideration for the function.
- Validation. How the result of the function is processed by the caller.
- Synchronization. Denotes the synchronous or asynchronous behaviour of the function.
- Initialization & termination. How the function is started and ended.
- Error-handling. Where detection and processing of errors must be done.
- Instance procedure. What is responsible for instantiation of the function.
- Monitoring. Special actionss in order to control the function when a problem occurs.
- Security. What are the security considerations applicable to the function.

3.2 Purpose

In this context, a "functional unit" is a set of functions gathered because they all apply to a common part of the development. The parser is a functional unit which interfaces with both the SRAM task and SROENV functional unit. The SROENV functional unit is the one already existing in the RSO subsystem and responsible for processing the start of the subsystem. At startup of RSO subsystem, the parser is responsible for building all the tables it needs for working on-line. At the time an /SD command is given for a device, the parser should build the work tables used during the parsing itself. The data to parse is given by the SRAM task. The parser is executed under control of the SRAM task, that is, on P1 level.

The following functions are included in the parser :

- Startup (PRSINIT). This function is responsible for building PRSDEV tables for all device-types. This table contains the "pattern" string used for detecting printer escape sequences to be stored. Only one copy of this table is needed for all parsers currently activated. The function PRSINIT is related to the SROENV function-unit.

- Start device (PRSSD). This function is called when a /SD command is given. At that time, the PRSIC table is built. This table is associated with only one device name, and has both read and write access.
- Parsing of data (PRSBUF). This is the main function of the parser. It receives a buffer containing the data to be sent to the printer. It uses pattern of the PRSDEV table to identify native escape sequence, and PRSIC table for internal work. The sequences which have been identified as needing to be saved are stored into a buffer associated to the PRSIC table. At the end of the parsing, all the sequences saved in the parser's internal table PRSIC are copied into a buffer (restart buffer), owned by SRAM, which will be saved along with checkpoint information and sent again to the printer in case of restart.
- Stop device (PRSSDNO). This function processes /SD ..., USE=NO command. It destroys the PRSIC table and releases the memory allocated to it.
- Clear context (PRSCLR). This function resets the context of the parser in case of the processing of a new job.

3.3 Realization

3.3.1 Control

The figure A.1 represents the global architecture of the parser and how its functions interface with the RSO subsystem.

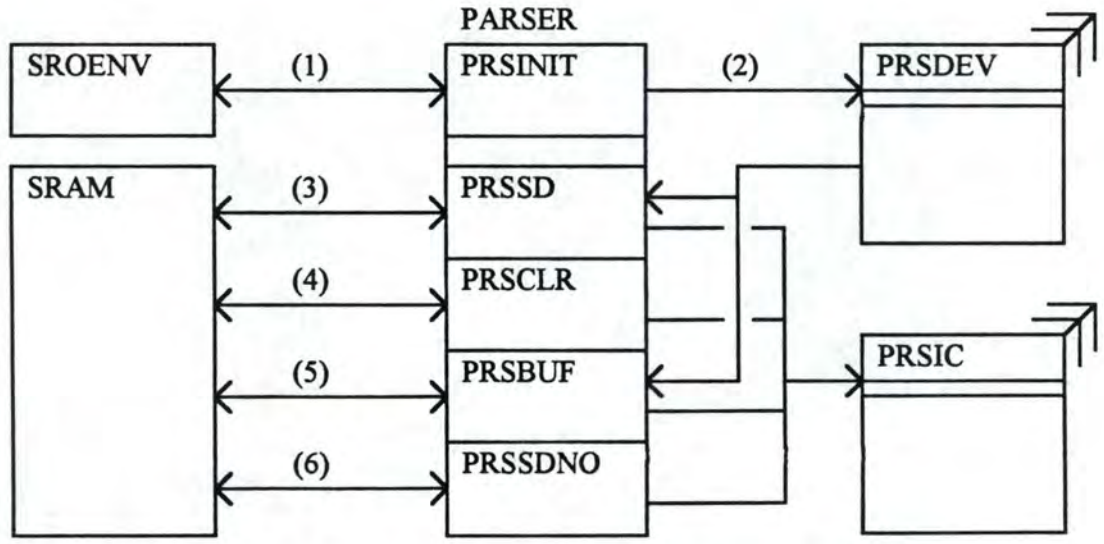


Figure 3.1 - Architecture of the parser

At startup of the RSO subsystem, SROENV calls the PARSER functional-unit for building all the PRSDEV tables in the SRAM memory pool.

The PRSDEV tables which are created at the startup time are set to read access and made available to all parsers. Each PRSDEV table contains patterns specific to one device/type.

On start of processing a new job, the function PRSCLR is called for clearing the old context.

At start-device, the parser is called for building work table PRSIC.

At parsing time, the parser is called by SRAM to build the restart buffer.

At stop-device, the parser destroys the PRSIC table and release memory.

3.3.2 Data

3.3.2.1 SRAM memory pool

The SRAM memory pool contains all the SRAM internal tables and the PARSER internal tables (PRSDEV) built from the RSO parameters file at the RSO subsystem startup. All these tables are created by the SROENV function-unit. This memory pool is common to all SRAM tasks, and so also to all PARSER.

3.3.2.2 Parser working table (PRSIC).

The table PRSIC has read/write access when processing /SD request. The memory size of the table is computed, and the memory is allocated. Then the table is initialized. During parsing, the table is used to store the sequence. When processing /SD ..., USE=NO, the table is destroyed and the

memory is released. The PRSIC table is located in the class-6 memory of the SRAM task. One copy of this table exist for every started device. This table is described with the function PRSSD. The table is made of two parts : one containing the fixed-length part of the table, and one containing variable-length part.

3.3.2.3Parser input buffer.

This is the buffer filled by SRAM and given to the parser as a parameter. This buffer contains data to be sent to the printer. It is accessed read only by the parser. The size of this buffer is function of SRAM. This buffer contains EBCDIC code, even when the printer accessed is pure ASCII. That leads to the fact that all the patterns in PRSDEV have to be in EBCDIC also.

3.3.2.4 Parser restart buffer (PRSRST).

This buffer is given to the parser by SRAM. It has no special structure, and contains the sequences stored by the parser in a form directly compatible with the printer. As the original buffer, the restart buffer contains EBCDIC code. The size of the buffer is determined by SRAM.

3.3.3 Assumptions

See functions description.

3.3.4 Performance

See functions description.

3.3.5 Validation

See functions description.

3.3.6 Synchronization

The interface between the SRAM task and the parser is synchronous. This is true also at the startup, when SROENV calls the parser for initialization.

3.3.7 Initialization and termination

3.3.7.1 Initialization

At startup of the RSO subsystem, all the tables in the SRAM memory pool are built. This is done by the PRSINIT function. The second level initialization (creation of PRSIC table) is done on /SD processing. It is processed by the PRSSD function. When a SRAM task is created, it does have to process a /SD command.

3.3.7.2 Termination

The parser is terminated when SRAM task is stopped, which occurs when the RSO subsystem is destroyed (STOP-SUBSYSTEM command), and also when the SRAM task has no more devices to manage (i.e. at the /SD ..., USE=NO command of the last printer managed by that SRAM task). The data structures created on subsystem initialization are destroyed with all the subsystem, and those created on /SD command are destroyed when a /SD ...,USE=NO command is processed.

3.3.8 Error-handling

All the errors that can occur are detected into the functions and returned to the caller. The errors that can occur at startup should be considered as important enough to cancel subsystem initialization. See functions description for more details.

3.3.9 Instance procedure

The function PRSINIT can only be requested by SROENV functional-unit. It is required that this function be completed before any call to parser itself. All the other functions are called by SRAM. The parser only uses data structures known by SRAM.

3.3.10 Monitoring

No monitoring function is pertinent in this context.

3.3.11 Security

The security problems are those of SRAM. The memory pool common to all SRAM tasks is protected with SCOPE=GROUP when created, which allows only task running under the TSOS userid to get access to it. Moreover, the memory pool is in read-only access for all SRAM tasks, and only accessible for writing by its creator, here the SROENV functional unit. Once created, the SRAM memory pool cannot be modified, unless deleting and restarting the RSO subsystem, thus insuring that during a session, it cannot be damaged by an error in an SRAM task.

3.4 Function PRSINIT

3.4.1 Purpose

This function builds the PRSDEV table needed by the parser for working. There are as many PRSDEV tables as device types supported by system. These tables resides in the SRAM memory pool common to all SRAM tasks.

3.4.2 Control

PRSINIT function accesses the RSO parameters file by means of SROENV functional unit. This function must process all the parsing records of type x'40' in that file. The structure of type x'40' record is given in figure 3.2.

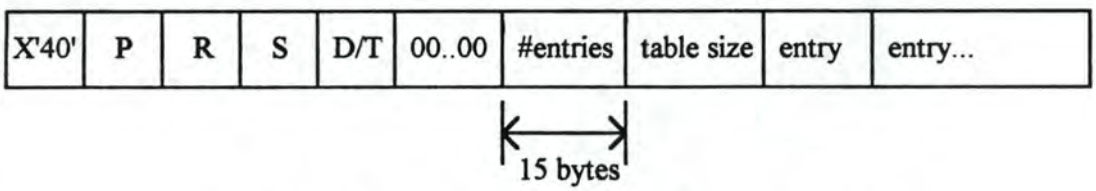


Figure 3.2 - Structure of record in RSO parameters file

Each entry contains all the information needed by the parser to process the buffer. The structure of an entry is given in figure 3.3.

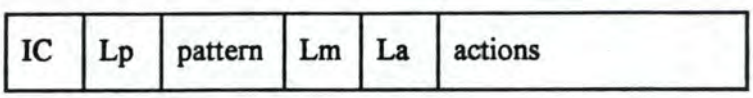


Figure 3.3 - Structure of the entry for one pattern

In this figure :

- IC is the internal code associated to the function described in pattern.
- Lp gives the length of the pattern in bytes. The pattern cannot be longer than 255 bytes. It has no sense to have a pattern length of 0.
- Lm is the maximum length of sequence to save for that internal code.
- La is the length of actions to do when processing that internal code.
- actions is the action string to do when processing the internal code.

Patterns and actions are described later in this document.

3.4.3 Data

In the SRAM memory pool, all PRSDEV tables are contiguous. The global structure of SRAM memory pool is given in figure 3.4.

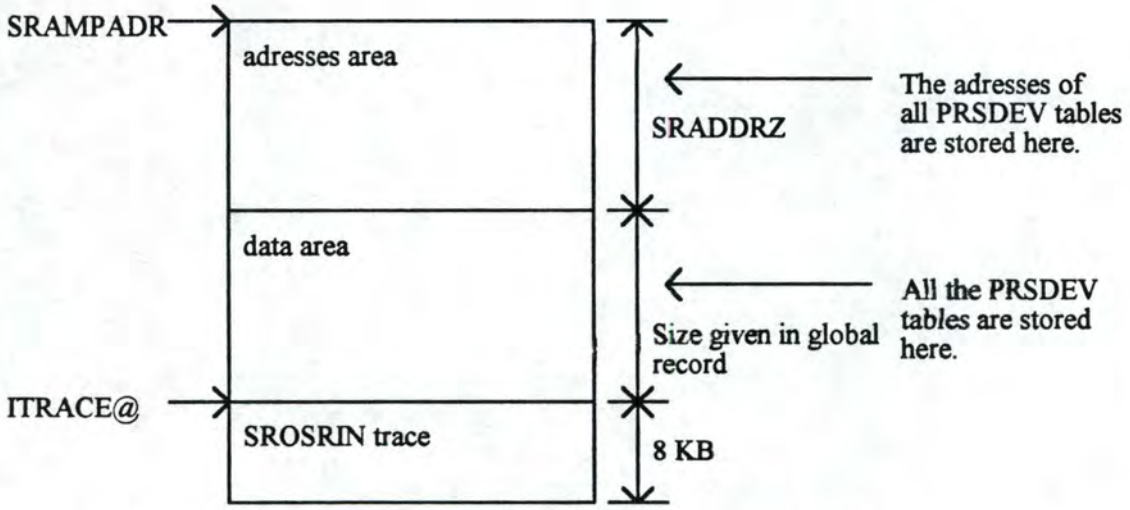


Figure 3.4 - Structure of the SRAM memory pool

In the data area, the structure of each PRSDEV table is split in two parts as given in figure 3.5.

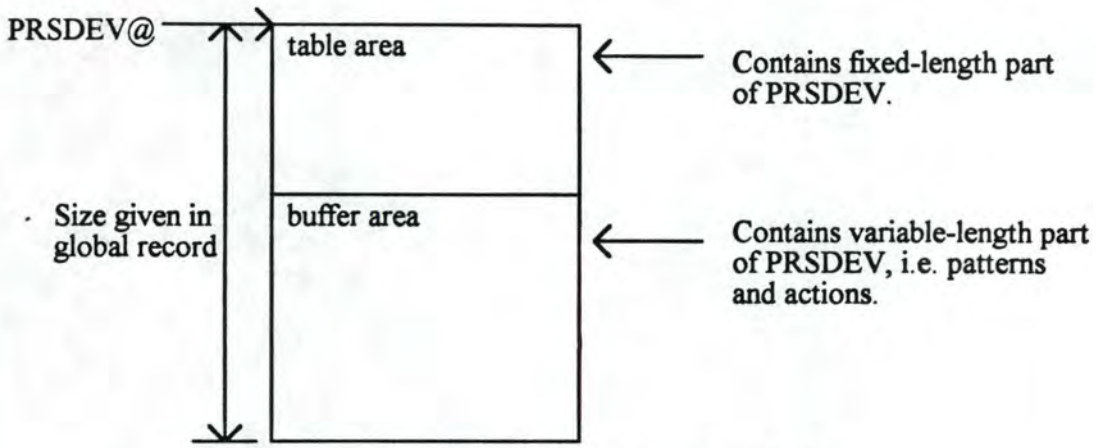


Figure 3.5 - Split structure of PRSDEV table

The patterns and actions are stored in the buffer area. PRSDEV tables are stored in the table area. The table area contains all fixed-length information about each pattern. The structure of the PRSDEV table is described in figure 3.6. The table describes, for each device-type supported, the sequences which are necessary to retain for restarting the printer after an error. These sequences are mainly resources-selection (font, CPI, LPI and so on) or resources-loading (such as tab positions). The sequences entered into this table are "pattern" with parameters, which during parsing are matched with the real values found in the data stream sent to the printer. This table is created in the memory pool of SRAM task, at subsystem initialization, by the function PRSINIT in SROENV functional-unit. This function is responsible for building all the resources needed for SRAM task to work.

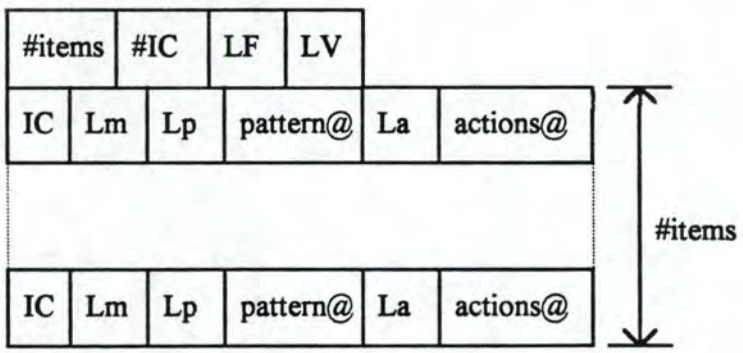


Figure 3.6 - Structure of PRSDEV table

The description of all items in the table is given in table 3.1.

Name	Length (bytes)	Description
#items	2	Contains the number of items in the table (1 to 65535).
#IC	1	Number of different internal codes.
LF	4	Length in byte of the fixed-length part of the PRSIC table
LV	4	Length in byte of the variable-length part of the PRSIC table
IC	1	Internal code used for parsing (0 to 255)
Lm	1	Maximal length of the sequence to save under this internal code
Lp	1	Length of pattern
pattern@	4	Address of pattern
La	1	Length of actions string
actions@	4	Address of actions string

Table 3.1 - Description of items in PRSDEV table

The actual values of pattern and actions are stored in a buffer in the format given in figure 3.7.

Figure 3.7 - Structure of the buffer containing patterns and actions

We choose this organization, rather than storing patterns and actions into the table, because patterns and actions are of variable-length, and that the size of the table should be kept as low as possible. This table is kept into the memory pool with read access. Each D/T supported is associated with one of these tables, regardless of the connection type. Each internal code is associated with one functionality of the printer. A given internal code can be repeated in the table, along with the pattern corresponding to the functionality of the printer. In that case, however, all the patterns associated to the same internal code should be different, and reflecting the differences existing in the data streams according to the connection type. The identifier of the table is the pattern, and not the internal code. Thus, one pattern is associated with one internal code, in a many-to-one relationship. All the occurences of PRSDEV tables should be referenced by a "central anchor table", as shown in figure 3.8. This structure is given as example, as the internal structure of the memory pool is under the responsability of SROENV functional unit, and not concerned by the parser.

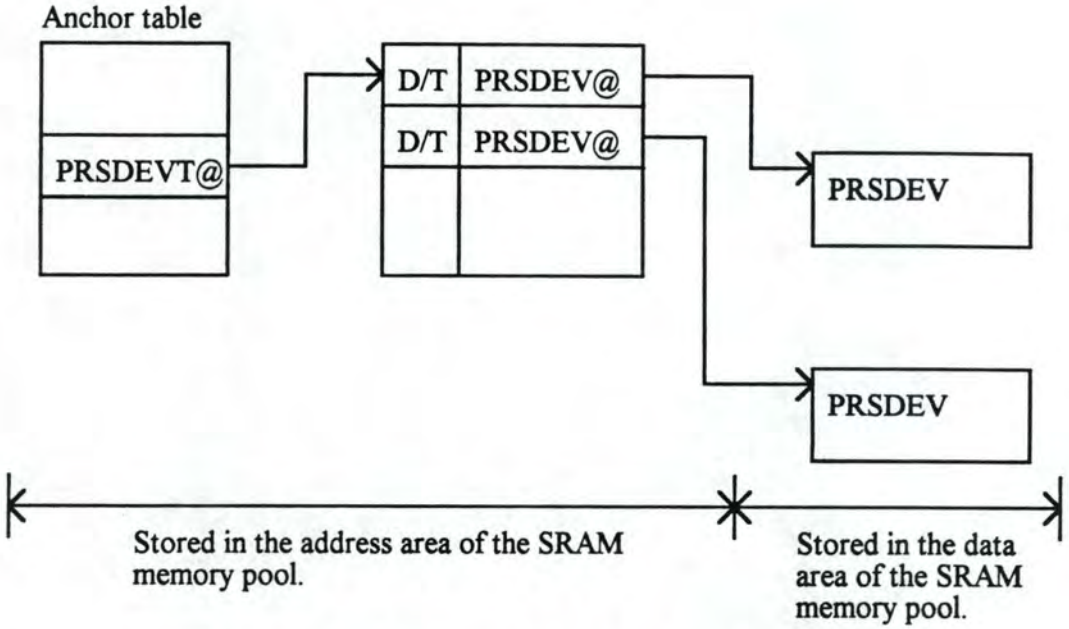


Figure 3.8 - Structure of the central anchor table

Important : for coherency reasons, if one IC is repeated in the table, all the Lm given for this IC should be the same. Lm is used to compute the size of buffer needed for storing the sequence (see PRSIC table). Other information which should be found in PRSDEV table (not yet completely defined) is the length of the PRSIC table.

3.4.3 Synchronization

The execution of this function is controlled by the caller. It is executed synchronously with it.

3.4.4 Initialization and termination

3.4.4.1 Initialization

The function is called by SROENV functional unit.

3.4.4.2 Termination

Upon completion, the function returns a code indicating possible errors.

3.4.5 Error-handling

Access errors on the RSO parameters file are processed by SROENV functional unit. When an inconsistency is found during the execution of PRSINIT function, the subsystem has to do without parsing for the whole session. All the size of table can be computed before call to PRSINIT. The only case of error that can be detected at

this stage are different maximum lengths for the same internal code. In that case, the largest number should be taken as reference.

3.4.6 Instance procedure

The PRSINIT function is executed under the control of the SROENV functional unit.

3.4.7 Monitoring

None.

3.5 Function PRSSD

3.5.1 Purpose

This function is part of the processing of a /SD command. Its main function is to build PRSIC table. This table is associated to a specific device name. The table has a read and write access. This function is executed in the SRAM processing of /SD command. When this function is completed, the parser is ready to start working for the device started.

The function should have access to the PRSDEV table corresponding to the device type of the /SD command. Basically, the function will process the PRSDEV table item by item, building an entry in the PRSIC table for each internal code detected. Associated with that internal code, the PRSIC table will contain a pointer to an area, in the buffer (variable-length area) of the PRSIC table, where the sequence should be saved.

3.5.2 Control

The interface of the function is as follow :

CALL PRSSD BUFFER RC

The size of the buffer is computed by information found in the PRSDEV table. When function ends, the buffer is filled with the PRSIC table and its associated buffer. The RC return code gives the result of the function, or an error code if a problem occured internally.

3.5.3 Data

The PRSIC table is built by this function. This table resides in the local SRAM memory. Before calling PRSSD, SRAM should allocate memory for PRSIC table (this is true for all functions of parser : they never allocate memory by themselves, but rather use memory allocated by the caller, and given as a parameter). For SRAM

to correctly allocate the memory, it must have access to the information giving table size. This information is stored in PRSDEV table. It can be computed during execution, but it is simpler and more efficient to store it into the RSO parameters file. As for PRSDEV table, the structure of the table is split into 2 parts : one index containing all fixed-length information, and a variable-length buffer containing actual values. The structure of table is given in figure 3.9.

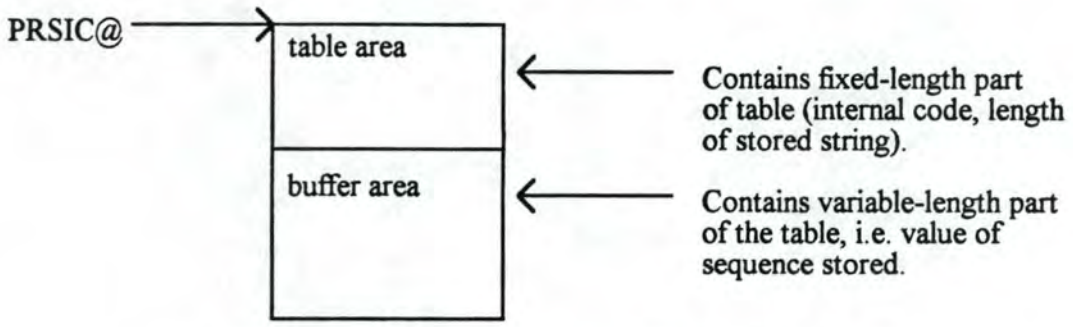


Figure 3.9 - Global structure of table PRSIC

The description of table follows. PRSIC is a table used by the parser for storing the native sequences of printer data stream recognized by the parser as needing to be saved. There is one occurence of that table by device-name. This table is allocated by the processing of /SD command, in the SRAM function unit. This table is detroyed with an /SD ...,USE=NO command, by function PRSSDNO. The table is made of two elements : the index table itself, and the buffer used to store the native sequences. The structure is given in figure 3.10.

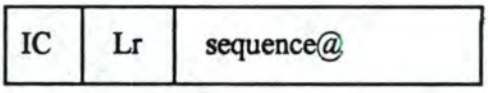


Figure 3.10 - Structure of one entry in PRSIC table

The table 3.2 describes in detail what we can find in an entry of the PRSIC table.

Name	Length (bytes)	Description
IC	1	Internal code of the function associated with the sequence
Lr	1	Length of the sequence saved in buffer or 0 if no sequence saved
sequence@	4	Address of place in buffer where the sequence should be saved

Table 3.2 - Description of the items in PRSIC table

The table size (including buffer) is given into the PRSDEV table. Structure of PRSIC table with its buffer is given in figure 3.12.

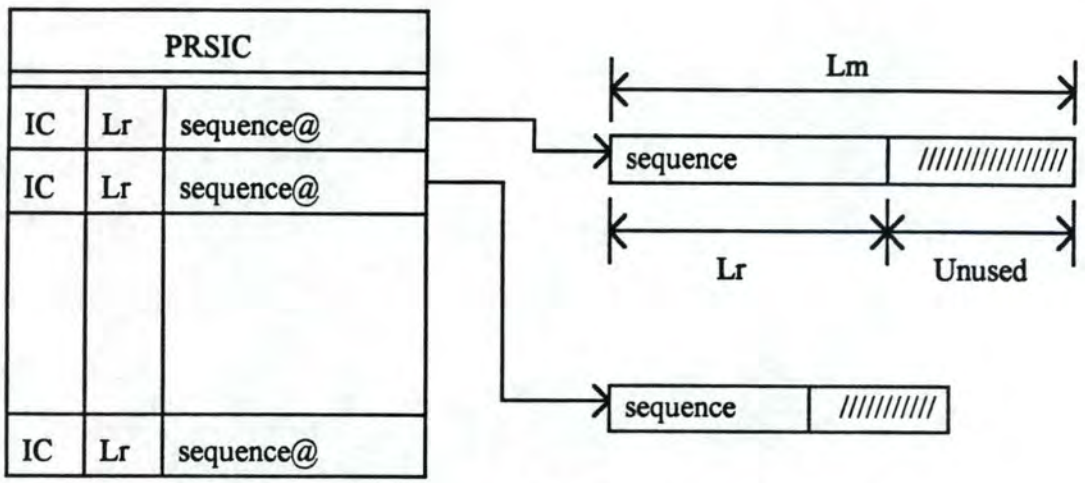


Figure 3.12 - Structure of PRSIC table and its buffers

The address of table is given as a parameter to the parser.

3.5.4 Synchronization

The PRSSD function is executed synchronously with and under the control of the SRAM task. This processing is done when REQU contigency is activated in the SRAM task.

3.5.5 Initialization & termination

3.5.5.1 Initialization

The function is initialized by SRAM.

3.5.5.2 Termination

The function terminates by returning a value indicating potential error.

3.5.6 Error-handling

All the errors detected into the function are returned to the caller (SRAM). Errors on the length of the table PRSIC are important enough to stop processing and directly returning, as they denote a coherency problem in the RSO parameters file. For example, allocating too little space for the table.

3.5.7 Instance procedure

The function is instancied by SRAM on call of it.

3.5.8 Monitoring

No special actions pertaining to monitoring is relevant in this context.

3.6 Function PRSBUF

3.6.1 Purpose

This is the main function of parser. This function is responsible for the parsing in itself, and for storing information into the PRSIC table. The function is called by SRAM just before starting on optional EBCDIC to ASCII conversion. The function must have access to several objects. These objects and the kind of access are given in table 3.3.

Object name	Access
Printer buffer	Read
PRSDEV table	Read
PRSIC table	Read/write
Restart buffer	Read/write

Table 3.3 - Object and access for function PRSBUF

3.6.2 Control

The execution of PRSBUF could be seen as a 3-steps process : first, initialization of data structure for restart ; second, processing byte by byte of the printer buffer, with storing of interesting parts ; and third, building of restart buffer. During the parsing, when detecting an escape sequence that should not be converted from EBCDIC to ASCII (that is, when the actions defined for that pattern contain a special code indicating "forced exit"), the parser should stop the processing, even in the middle of the buffer currently processed, and should return the following information :

- Displacement in the buffer where the EBCDIC to ASCII conversion should be turned off
- Number of bytes for which the conversion should be stopped.

When SRAM gets these information from the parser, it should call the parser again, for processing of the remaining of the buffer. In that case, a single buffer in SRAM could lead to several calls to the parser, as many as there are escape sequences

which should not be converted from EBCDIC to ASCII. These kinds of escape sequences are mainly graphic bitmaps and font loading. The processing of PRSBUF function of the parser is described in the logic flow diagram pictured in figure 3.14.

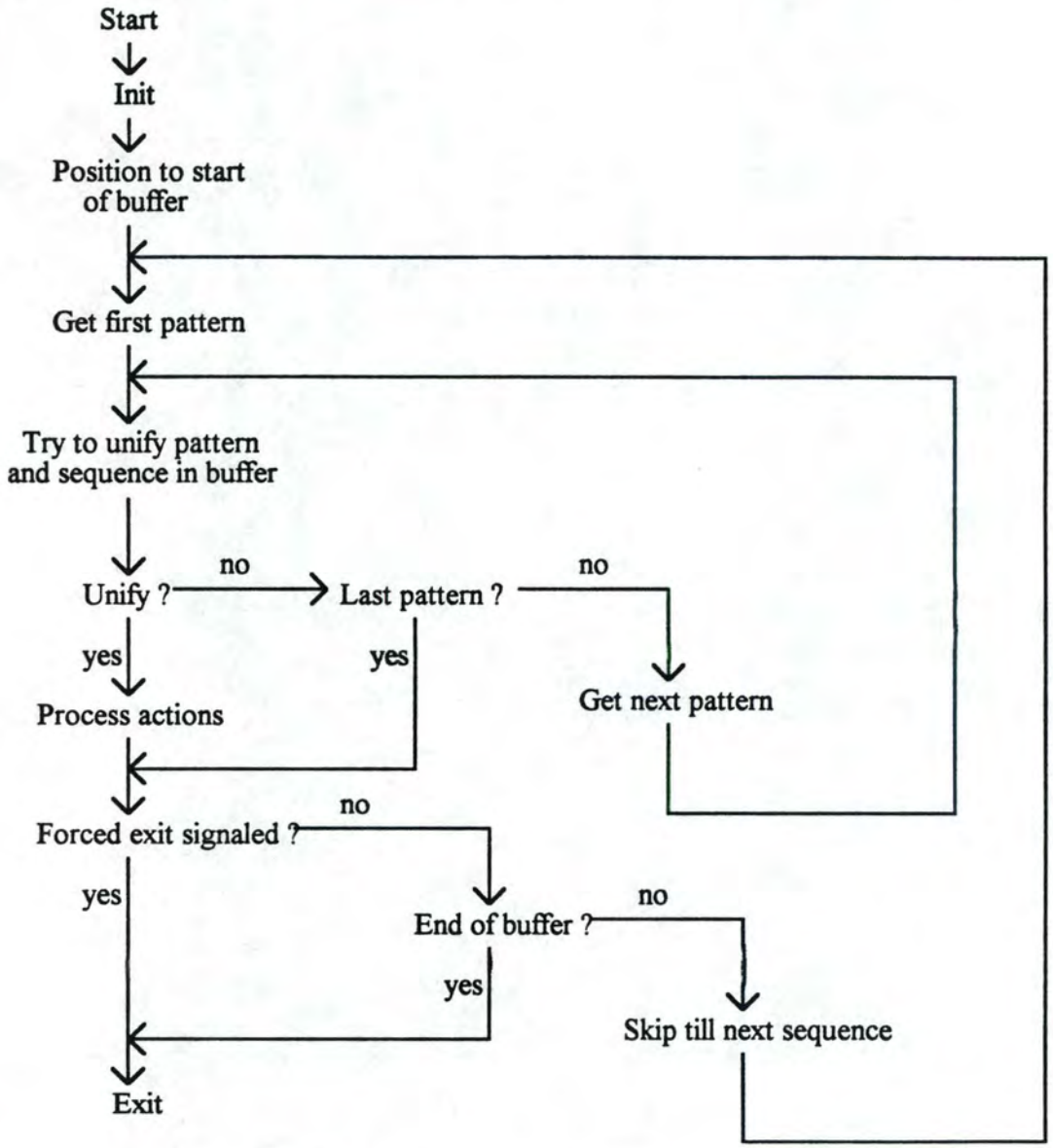


Figure 3.14 - Logic flow for function PRSBUF

3.6.3 Data

The tables PRSDEV and PRSIC are described under the functions building them (i.e. PRSINIT and PRSSD).

3.6.3.1 Printer buffer

It is not structured in a special way for the parser, but rather formatted for the specific printer addressed. Its size is computed by SRAM.

3.6.3.2 Restart buffer

Strutured in the same way that printer buffer. Its size is stored in the RSO parameters file, and can also be found in the PRSDEV table. The restart buffer is allocated by SRAM, function PRSSD, before calling the parser. The restart buffer should be stored along with checkpoint informations.

3.6.4 Synchronization

The processing of the parser is completely synchronous with the SRAM task.

3.6.5 Initialization & termination

3.6.5.1 Initialization

The parser should be notified when a new job is ready to be dispatched. At that moment, sequences stored from preceeding job should be cleared in order to keep the printer in the state asked for by the user.

3.6.5.2 Termination

On termination, the restart buffer will be filled with sequences to send again to reload resources into printer. Attention should be paid to the order in which the native sequences are copied into the restart buffer. Normally, they must be stored in the order in which they appear in the user file, since some functionalities of the printer can be reset by a specific native sequence. Where to store this buffer is on SRAM's responsability. In a symmetric way from Init, the parser should be informed that a print job is ended (for example, to notify that an escape sequence has been cut in the middle).

3.6.6 Error-handling

All the errors that occur during parsing must be detected, and signaled to the caller (SRAM). The main problem that can occur is when trying to store, in the PRSIC table, escape sequences which are too long for the place allocated. In fact, this is an error in the RSO parameters file, but of course it should be detected during parsing.

Essentially, the parser should follow a "most conservative approach", that is, in case of problems, it should signal it, but also should exit in a "safe state" : data stored should be coherent and even if some functionalities are not saved due to this error, keep in mind that the parser activity is essentially for restart. It must not penalyze too much performance when no problems are encountered with the printer.

3.6.7 Instance procedure

The function is fully executed under the SRAM process.

3.6.8 Monitoring

None.

3.7 Function PRSSDNO

The function PRSSDNO is symmetrical to the function PRSSD. It consists in the processing of a /SD ...,USE=NO command. As the allocation of memory for PRSIC table is done in SRAM, the deallocation should be done in SRAM too. In fact, it seems that no operation should be done by the parser in that case.

3.8 Function PRSCLR

3.8.1 Purpose

On start of a new job, the parser should be notified to clear the old context" saved from previous job. The function PRSCLR does that, by clearing the information stored in PRSIC table.

3.8.2 Control

This function is essentially built to restore the PRSIC table as it is built by the PRSSD function. For each internal code in this table, the function should indicate a null length for the value stored in the buffer part of PRSIC. In fact, this function will be implemented in the PRSBUF function. A special switch in the parameters will signal that this is the first buffer of a new job, and so that clearing should be done.

3.8.3 Data

This function use PRSIC table described along with PRSSD function.

3.8.4 Synchronization

All the process is done under the control of SRAM task, synchronously.

3.8.5Initialization & termination

3.8.5.1 Initialization

Done by the SRAM task on processing of the first buffer of a new job.

3.8.5.2 Termination

On termination, control is returned to SRAM.

3.8.6 Error-handling

Only access error on the table PRSIC, which must be read/write, can be detected in this function.

3.8.7 Instance procedure

This function is used only in case of the start of a new job on the printer.

3.8.8 Monitoring

No action for monitoring in this function.

3.9 Language description

3.9.1 Pattern language

3.9.1.1 Objectives

The language described here is the one used in the expression of pattern. As it is based on Xprint implementation, it uses a structure derived from the format specification of scanf function in C language.

For the description of escape sequence, extensions to the C language have been made. The same "escape character" (the percent sign %) is used, but with other codes. In this language, the characters which are part of the sequence appear directly into the pattern, and the parameters (or repetition of parameters) appear with a % sign as prefix.

3.9.1.2 Language description

The pattern is a string of variable-length. The matching should be done from left to right, on a character-by-character basis. When encountering a "%" sign, the characters following it indicate the type of parameter to parse, and optionally its length. The table 3.3 lists the codes used.

Code	Meaning
%[n]d	Decimal value of n digits maximum
%[n]s	String of n characters maximum. MUST Be followed by a character indicating the delimiter of the string
%[n]l	Same as %d, but this information is a length, generally used to indicate the length of the sequence
%[n]c	Exactly n characters of lowercase (n=1 by default)
%[n]C	Exactly n characters of UPPERCASE (n=1 by default)
%[n]a	Exactly n characters of any case (n=1 by default)
%%	The percent sign (only one occurence in the sequence)
%{	Beginning of repetition of pattern
%}	End of repetition of pattern

Table 3.3 - Pattern language

In all the codes where [n] appears, it indicates that the length information can be omitted. When omitted in the %d, %s, %l codes, the value of n is ignored, that is variable-length data is assumed. Special attention should be paid to the %s code. It must be followed by a character, which cannot appear into the string, and that will be used as the "string terminator". If not specified (that is, %s at the end of the pattern), unpredictable results can occur. It is STRONGLY recommended to use the optional length in the codes ([n]). If it is not used, performances can be greatly degraded, as the parsing can be very long.

Example : the pattern

```
ESC [%d;%dH
```

match the sequence

```
ESC [1;24H .
```

In the RENO language (a proprietary language used by printers from Siemens-Nixdorf), font selection is as follow :

```
\FO'COURIER.8.N.1';
```

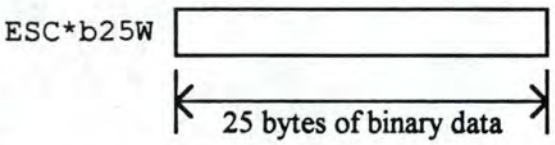
The pattern should be :

```
\FO'%s';
```

and the first quote (') encountered during the parsing of %s will be used as the string terminator.

The %l code should only occurs once, as there is no need for giving two different lengths in a single sequence.

For example, the sequence for bitmap in PCL is :



The pattern :

```
ESC *b%lW
```

will catch that case, and special action code will allow the parser to skip the 25 bytes following the sequence (and signaling to caller that the EBCDIC to ASCII conversion should be disabled for that length).

3.9.2 Action language

3.9.2.1 Objectives

As soon as a given sequence is matched with a pattern, the action codes associated with this pattern are interpreted. In these action codes, one can specify whether or not the sequence should be saved, if some IC should be reset, to skip n bytes after the sequence, and so on. It is also in the action codes that one signals that processing should be stopped in order to by-pass EBCDIC to ASCII conversion.

3.9.2.2 Language description

The actions are coded on one byte. This byte is followed by 0 to n bytes of parameters, depending on the action code. The interpretation of actions is done from left to right until the end of the string. The following table give action codes, their parameters and their meaning.

Action code	Parameters name (length in bytes)	Description
X'01'	IC (1)	Store the string under the given internal code.
X'02'	IC (1)	Reset the string stored under the given internal code.
X'03'	Number (1)	Skip the <number> of bytes after the sequence.
X'04'		Skip after the sequence, the number of bytes given by the %l code in the pattern.
X'05'	Length (1) Expression (n)	Store the evaluated expression that follows.
X'06'	Expression ID (1)	Store the evaluated expression guven by ID.
X'07'		Terminate processing and return the value associated with the %l code in pattern (forced exit).

Table 3.4 - Description of action language

The concept of "expression" introduced here, is only intended for space saving. It consists of associating an ID with a string used by the parser for storing value. These strings are defined in Xprint, under the term "reverse polish notation expression". The function doing the interpretation of these expression can largely be taken from Xprint module.

Example : the action code string that follows :

02 09 04

results in storing the sequence in the internal code 08, then resetting the internal code 09, and skipping the bytes following the sequence, for a length as parsed by the %l code in the pattern.

Chapter 4
Using filters to convert printing languages

4.1 Motivation

The fast evolution of printing technologies and consequently languages, makes it difficult for developers to modify applications to support these new languages. A simple, old and well-known printing language like the one used by Siemens-Nixdorf High-Performance printers will remain widely used for several years probably. But, preventing application written for this language to print on other printers, with different languages, is not admissible for the users. Moreover, the technology used by these printers is rather old, and not competitive with new technologies available now. The day will come where these printers will disappear, leaving users with applications no longer able to print a single line without modification.

The topic discussed here is about designing a filter for the High Performance printers from Siemens-Nixdorf, which converts the file designed for printing on these printers to a file formatted for PCL printers. It covers converting the user file, with all the resources that can be activated by the user with the /PRINT-FILE command.

4.2 The file format for High Performance printers

The resources for that kind of printers are the following :

- Character fonts. The High-Performance printers (HP-printers) always needs to be loaded with the character fonts asked for by the user. The printer has no resident font, but memory for up to 64 fonts of 256 characters each. The characters are bitmaps, defined in an 40 by 40 pixels matrix. A simple calculus shows that if you want to load all the fonts, you have to sent about 3 MB of information to the printer to load the fonts, and that before the job can start.
- Loop. The loop is loaded in the printer, where it will control the paper advance according to the control codes sent to the printer by the SPOOL.
- Electronic Form Overlay (EFO). The EFO is a bitmap description of the printing elements that can be added to each printed page. The printing of the EFO is controlled by the user for every page. The can be as large as 13 by 13 inches. This leads to about 1 MB of data to load the overlay.

For enabling the user to activate these resources, a special file format has been developed. This file format is record-oriented, with the characteristic that the first character of each line is never printed. This character is the "paper advance" indication for SPOOL. For example, a character of X'01' (01 in hexadecimal) indicate a single line step. The actual advance depend on the loop loaded, and given in fraction of inches (1/6, 1/8 or 1/12 of inch usually, but other size are available).

To indicate the beginning of the page, a record with the character X'81' or X'C1' should be encountered. Following this characters are 10 control characters, each one controlling the activation of a special characteristic of the printer. Among these controls are : the number of copies for the page, whether or not activating the EFO for this page, whether or not interpreting control codes embedded in the text, and so on.

Embedded in the text are control characters. These ones are never printed, but rather give the indication to switch to another font in the line.

4.3 Conversion of HP language to PCL

For converting HP-formatted files (that is, files which contains the control codes described before) to PCL, the filter a primary function : converting the record-oriented approach of HP-files to stream-oriented data flow of PCL. This conversion is done through the use of the stream functions available with the C language under BS2000.

Next, the main problem is to process the control codes at the top of the page. Whatever the case, detecting an X'81' or X'C1' character at the first position in the record results in a page feed in PCL. But the other control codes are to be simulated by the filter. For example, the indication of the number of copy to produce should be translated into the corresponding escape sequence in PCL. For each control character, one have to send, as necessary, the escape sequence corresponding.

The filter is only designed to convert user files. The resources specified in the /PRINT-FILE command are to be converted separately, and recalled when the print job is about to be started.

The filter has been designed as "off-line". Conversion of the HP-file to PCL must be done before starting the printing job. Then, the resulting file can be printed on any PCL printer. The figure 4.1 shows the flow of data when converting and printing.

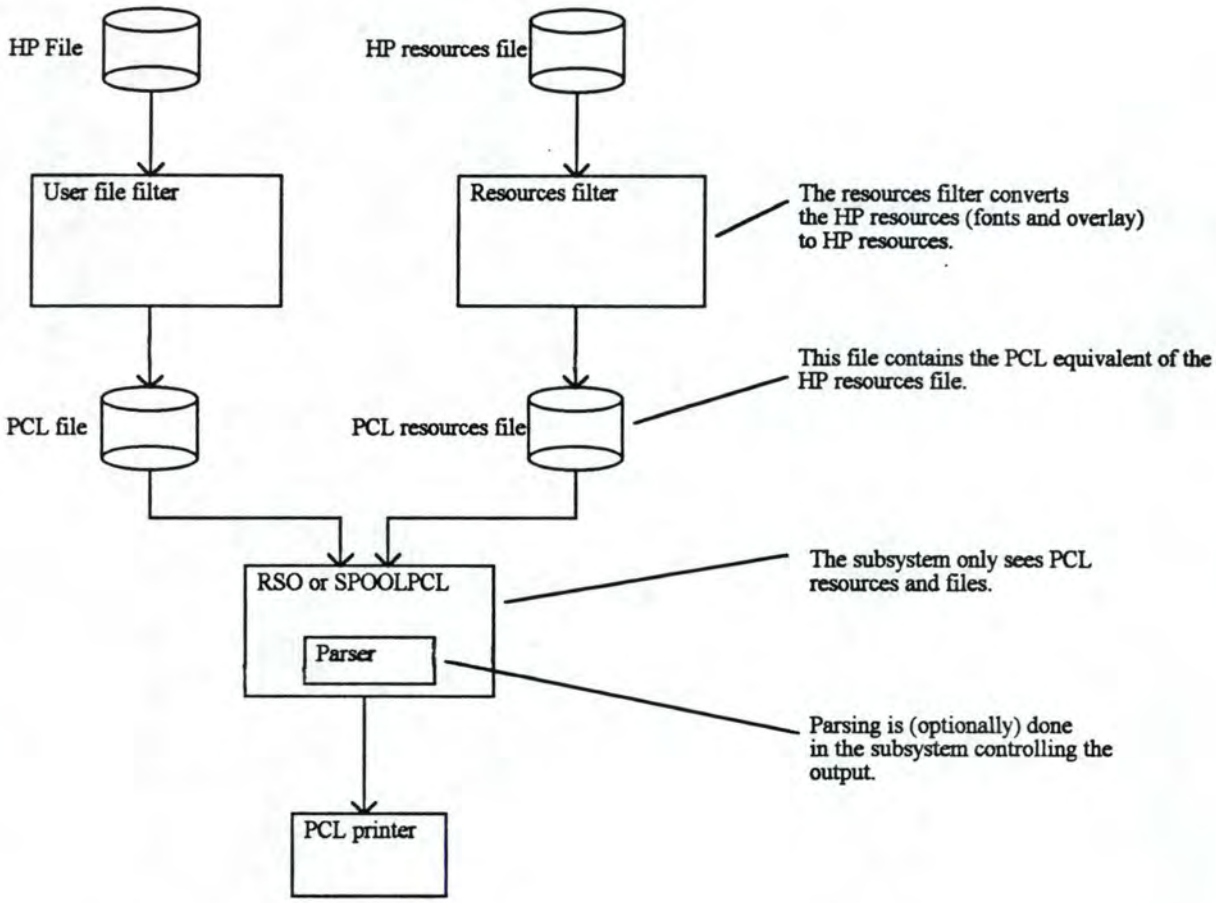


Figure 4.1 - Flow of data in the filter

The filter discussed here is only responsible for converting the user file. For resources, another filter has to be developed, which was not the objective here (In fact, the conversion of resources directly from the HP-format is not very easy. A better approach is to convert from the user specification, directly into the PCL resources file).

4.4 Integration of the filter into SPOOL subsystem

Up to know, we only introduced the "off line" approach of the filter. There is another solution : the user file filter can be integrated into the SPOOL subsystem. The resources have still to be converted "off line". A special feature of the SPOOL, called "system exit", give the user the opportunity to make a special processing of the data before it is sent to the printer. In the case of the SPOOL subsystem, the system exit 90 is activated just after reading a record from the user file. By adhering to the specifications of the system exit, the filter can then process the record just read, translating the HP-formatted record to a line containing only PCL. The only exception is for the first character : it is processed by the SPOOL controller to send the appropriate commands to the printer. When supporting PCL printer, this part of

the controller should be modified, for sending PCL sequences to the printer in place of HP-printer commands.

This architecture allowed minor modification to the SPOOL controller in order to support PCL printer.

Conclusion

The training period in Siemens-Nixdorf Software in Namur was a pleasant and interesting experience. This was the occasion to be involved in a large development and to get training in this field of computer science. We had to deeply analyzed technical information and hardware specification in order to understand the effective problem.

The development of the parser was a essential requirement for managing large print jobs. Integrated into the problem of restart, it needed lots of work to maintain coherency into the whole subsystem RSO. The problem is not completely solved, however. The development of new technologies is still important, and will need stronger systems in order to correctly support them. A first sight, printing problems seems to be rather simple and well-managed. But this is not the case : usage of these new technologies by the user requesting more and more functionalities leads to more and more complex systems. It seems that new standards are needed for providing more consistent and powerful interfaces to the printing environment. That would be the occasion, for the computing environment, to develop new methodologies, giving printing a new place in the computing systems.

Bibliography

[1] PostScript Language Reference Manual, Second Edition ; Adobe Systems Incorporated ; Addison-Wesley Publishing Company, Inc. ; 1991

[2] Intelligent Printer Data Stream Reference ; International Business Machines Corporation ; 1991 ; reference S544-3417-03.

[3] PCL 5 Reference Manual ; Hewlett-Packard.

[4] Remote Spool Output Manuals ; Siemens-Nixdorf.

[5] SPOOL Manuals ; Siemens-Nixdorf.

[6] TCP/IP Tutorial and Technical Overview ; International Business Machines Corporation ; 1992 ; reference GG24-3376-03.

Table of contents